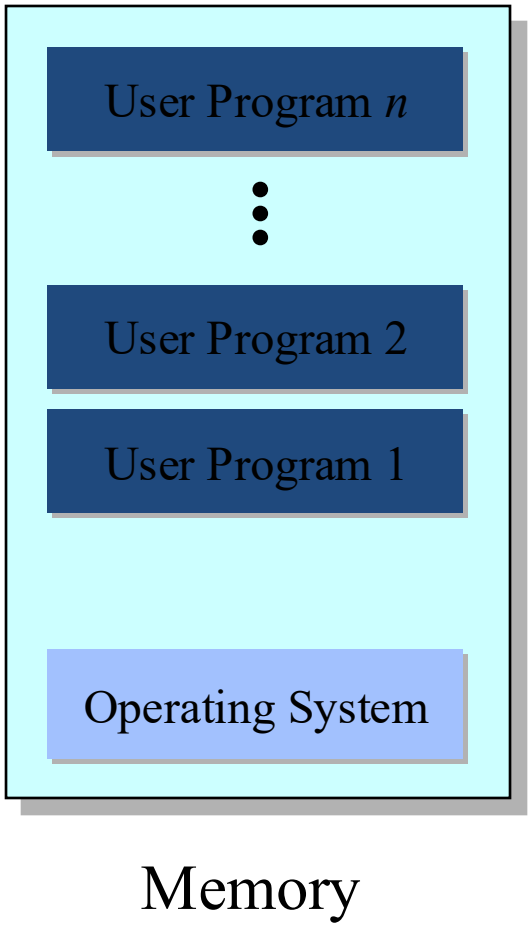
Page Replacement Algorithms

Don Porter

Portions courtesy Emmett Witchel and Kevin Jeffay

Virtual Memory Management: Recap

- Key concept: Demand paging
  - Load pages into memory only when a page fault occurs
- Issues:
  - Placement strategies
    - Place pages anywhere – no placement policy required
  - Replacement strategies
    - What to do when there exist more jobs than can fit in memory
  - Load control strategies
    - Determining how many jobs can be in memory at one time



Memory

Page Replacement Algorithms

- Typically $\sum_i VAS_i$ >> *Physical Memory*
- With demand paging, physical memory fills quickly
- When a process faults & memory is full, some page must be swapped out
  - Handling a page fault now requires ***2*** disk accesses not 1!

Which page should be replaced?

Local replacement — Replace a page of the faulting process
Global replacement — Possibly replace the page of another process

Page Replacement: Eval. Methodology

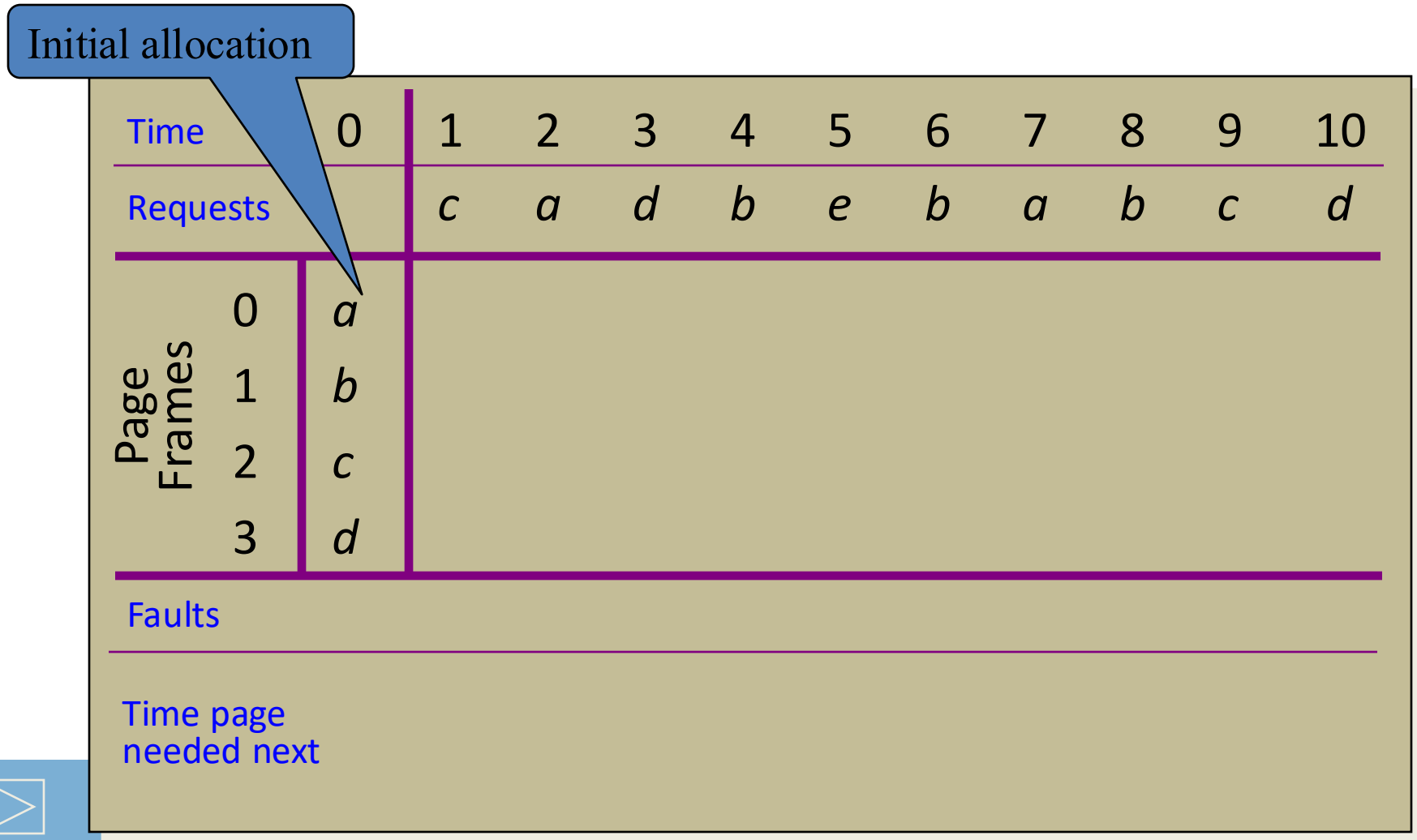
- Record a *trace* of the pages accessed by a process
  - Example: (Virtual page, offset) address trace...
    (3,0), (1,9), (4,1), (2,1), (5,3), (2,0), (1,9), (2,4), (3,1), (4,8)
  - generates page trace
    3, 1, 4, 2, 5, 2, 1, 2, 3, 4 (represented as *c*, *a*, *d*, *b*, *e*, *b*, *a*, *b*, *c*, *d*)
- Hardware can tell OS when a new page is loaded into the TLB
  - Set a used bit in the page table entry
  - Increment or shift a register

Simulate the behavior of a page replacement algorithm on the trace and record the number of page faults generated

fewer faults ⇒ *better performance*

Optimal Strategy: Clairvoyant Replacement

- Replace the page that won't be needed for the longest time in the future



Optimal Strategy: Clairvoyant Replacement

- Replace the page that won't be needed for the longest time in the future
- Also called Belady's MIN algorithm (cuz it MINimizes swapping!)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	*d*	*d*	*d*	*d*	*d*	***e***	*e*	*e*	*e*	*e*	*e*
Faults							•					•

Time page needed next

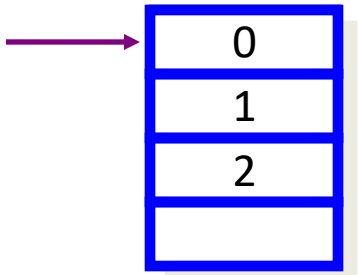
- At time 5: *a* = 7, *b* = 6, *c* = 9, *d* = 10
- At time 10: *a* = 15, *b* = 11, *c* = 13, *d* = 14

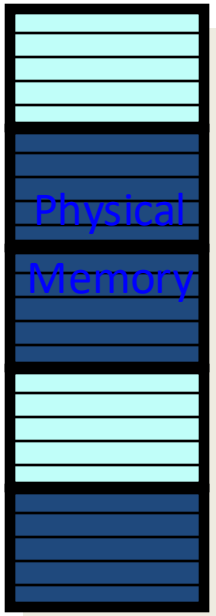
Where on the motherboard is my crystal ball?

- Hint: it isn't there
- So we have to use our knowledge at each point in time
  - Technical keyword: online algorithm

Local Replacement: FIFO

- Simple to implement
  - A single pointer suffices
- Performance with 4 page frames:

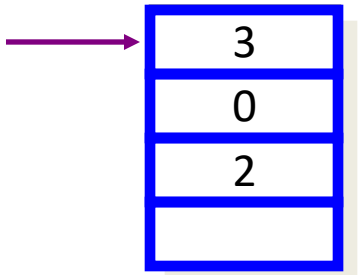


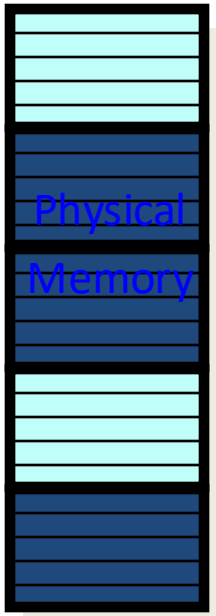


Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
Faults												

Local Replacment: FIFO

- Simple to implement
  - A single pointer suffices
- Performance with 4 page frames:





Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	**e**	e	e	e	e	**d**
	1	b	b	b	b	b	b	b	**a**	a	a	a
	2	c	c	c	c	c	c	c	c	**b**	b	b
	3	d	d	d	d	d	d	d	d	d	**c**	c
Faults							•		•	•	•	•

Least Recently Used (LRU) Replacement

- Use the recent past as a predictor of the near future
- Replace the page that hasn't been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
Faults												
Time page last used												

Least Recently Used (LRU) Replacement

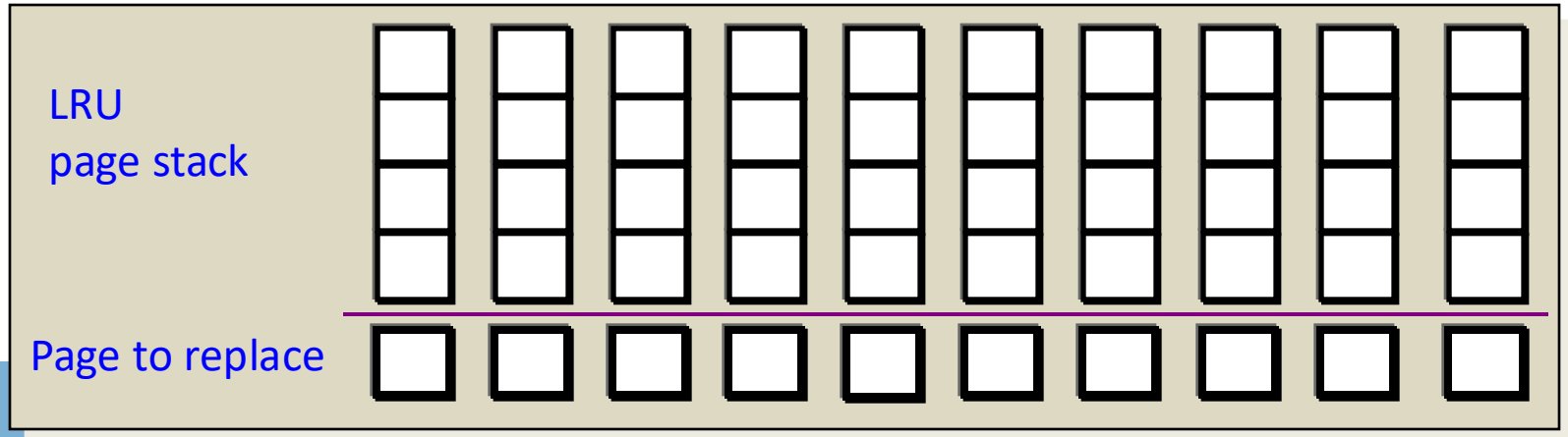
- Use the recent past as a predictor of the near future
- Replace the page that hasn't been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•
Time page last used						*a* = 2 *b* = 4 *c* = 1 *d* = 3				*a* = 7 *b* = 8 *e* = 5 *d* = 3	*a* = 7 *b* = 8 *e* = 5 *c* = 9	

How to Implement LRU?

- Maintain a "stack" of recently used pages

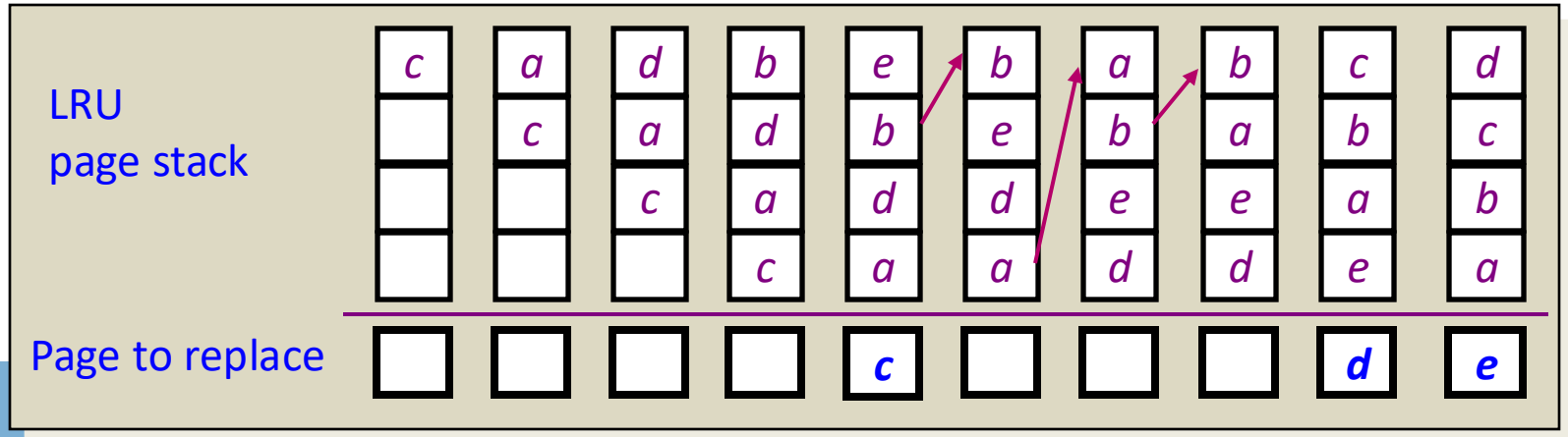
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•



How to Implement LRU?

- Maintain a "stack" of recently used pages

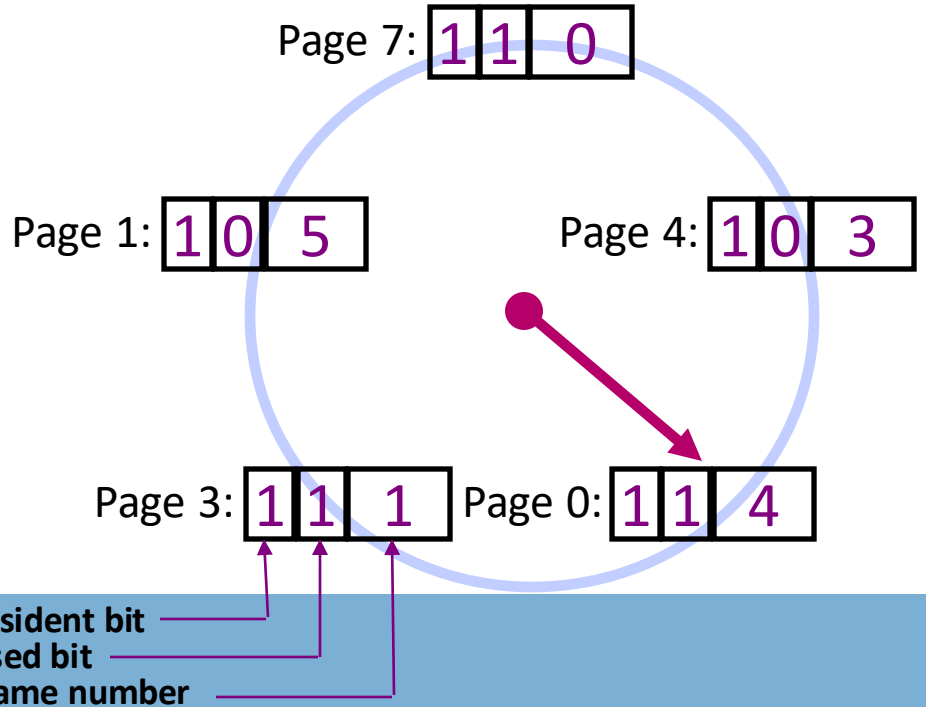
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							●				●	●



- What is the goal of a page replacement algorithm?
  - A. Make life easier for OS implementer
  - B. Reduce the number of page faults
  - C. Reduce the penalty for page faults when they occur
  - D. Minimize CPU time of algorithm

Approximate LRU: The Clock Algorithm

- Maintain a circular list of pages resident in memory
  - Use a *clock* (or *used/referenced*) bit to track how often a page is accessed
  - The bit is set whenever a page is referenced
- Clock hand sweeps over pages looking for one with *used* bit = 0
  - Replace pages that haven't been referenced for one complete revolution of the clock



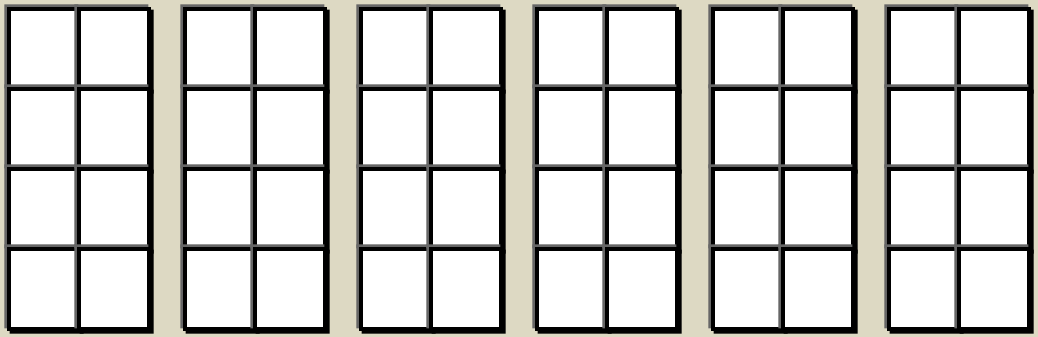

```
func Clock_Replacement
begin
   while (victim page not found) do
      if (used bit for current page = 0) then
         replace current page
      else
         reset used bit
      end if
      advance clock pointer
   end while
end Clock_Replacement
```

Clock Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*						
	1	*b*	*b*	*b*	*b*	*b*						
	2	*c*	*c*	*c*	*c*	*c*						
	3	*d*	*d*	*d*	*d*	*d*						
Faults												

Page table entries for resident pages:

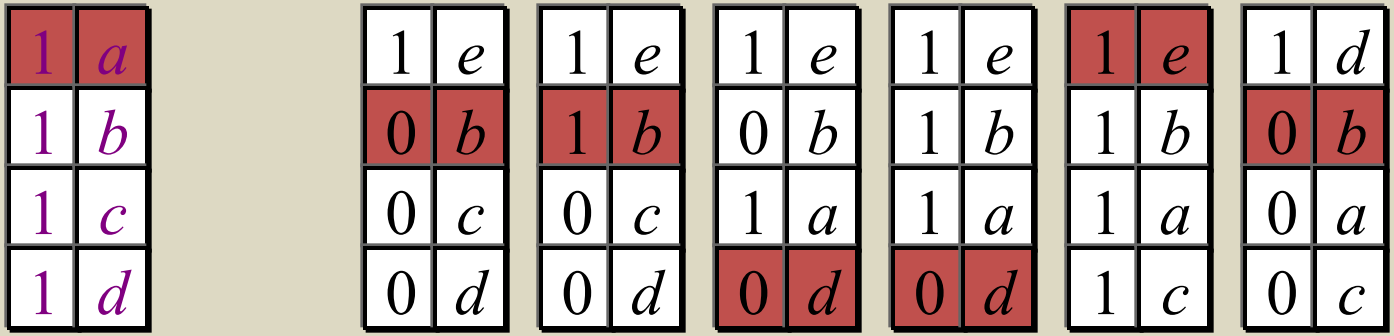
1	*a*
1	*b*
1	*c*
1	*d*



Clock Example

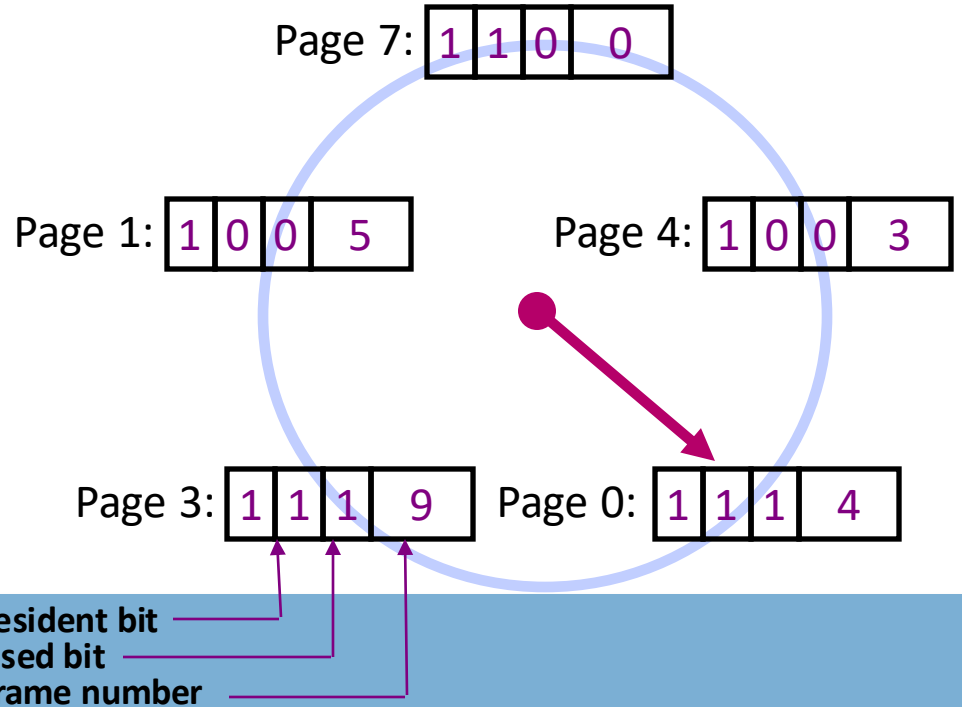
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	**e**	e	e	e	e	**d**
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	**a**	a	a	a
	3	d	d	d	d	d	d	d	d	d	**c**	c
Faults							•		•		•	•

Page table entries for resident pages:



Optimization: Second Chance Algorithm

- There is a significant cost to replacing "dirty" pages
  - Why?
    - Must write back contents to disk before freeing!
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the *dirty bit* and the *used bit* to drive replacement



Second Chance Algorithm

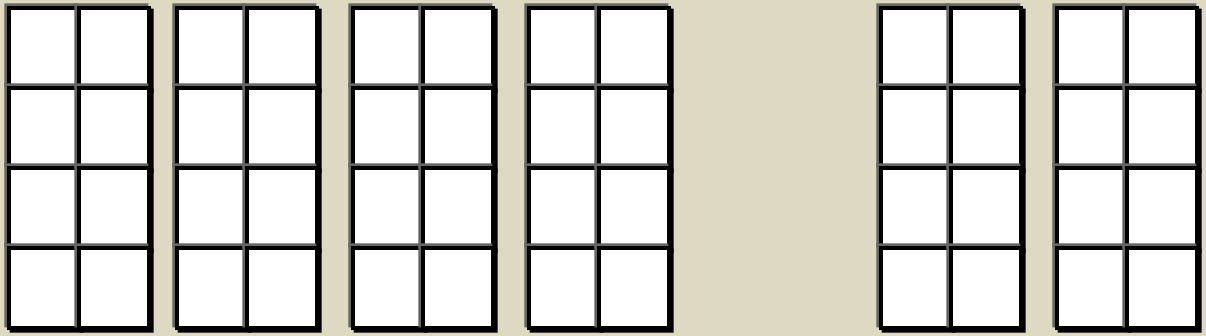
Before clock sweep		After clock sweep	
used	*dirty*	*used*	*dirty*
		replace page	
0	0	0	0
0	1	0	0
1	0	0	1
1	1		

Second Chance Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Faults												

Page table entries for resident pages:

10	a
10	b
10	c
10	d



Second Chance Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	***d***
	2	c	c	c	c	c	***e***	e	e	e	e	e
	3	d	d	d	d	d	d	d	d	d	***c***	c
Faults							•				•	•

Page table entries for resident pages:

10	a	11	a	00	a^*	00	a	11	a		11	a		00	a^*	
10	b	11	b	00	b^*	10	b	10	b		10	b		10	d	
10	c	10	c	10	e	10	e	10	e		10	e		00	e	
10	d	10	d	00	d	00	d	00	d		10	c		00	c	

Local vs. Global Replacement

- The examples to date assume either:
  - Only one process on the system (global, but facile), or
  - One process that has only 4 page frames (local)
- Let's now consider the issue of when to take memory away from one process and give it to another
  - Truly global page replacement

- Our goal is still to minimize page faults — system wide
- Let's start by considering whether to take memory away from a process…

Local Replacement and Memory Sensitivity

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames													
0	*a*												
1	*b*												
2	*c*												
Faults													

Page Frames													
0	*a*												
1	*b*												
2	*c*												
3	–												
Faults													

Local Replacement and Memory Sensitivity

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames														
	0	*a*	*a*	*a*	*a*	***d***	*d*	*d*	***c***	*c*	*c*	***b***	*b*	*b*
	1	*b*	*b*	*b*	*b*	*b*	***a***	*a*	*a*	***d***	*d*	*d*	***c***	*c*
	2	*c*	*c*	*c*	*c*	*c*	*c*	***b***	*b*	*b*	***a***	*a*	*a*	***d***
Faults						•	•	•	•	•	•	•	•	•

Page Frames														
	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	–			***d***	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*
Faults					•									

Key observations

- Decreasing the number of page frames for a process may increase swapping
  - But it isn't a linear relationship
- There is a low-water mark for every process where it goes from rarely swapping to constantly swapping
  - Goal 1: Don't let a process stay below this line
- And some situations where a process is not actually using all of its page frames
  - Goal 2: Find those cases and reclaim memory that won't be missed
- But how do we know?
  - Let's start by revisiting the clairvoyant approach

Optimal Replacement with a Variable Number of Frames

- *VMIN* — Replace a page that is not referenced in the *next* τ accesses
- Example: $\tau = 4$

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	• $t = 0$										
Page *b*	-										
Page *c*	-										
Page *d*	• $t = -1$										
Page *e*	-										
Faults											

Optimal Replacement with a Variable Number of Frames

- *VMIN* — Replace a page if not referenced in the *next* τ accesses
  - Not necessarily just on a page fault (can also happen when waiting/blocked)
  - Calculate on each step, even if no fault

- Example: $\tau = 4$

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	c	d	b	c	e	c	e	a	d
Pages in Memory: Page *a*	• *t* = 0	-	-	-	-	-	-	-	-	F	-
Page *b*	-	-	-	-	F	-	-	-	-	-	-
Page *c*	-	F	•	•	•	•	•	•	-	-	-
Page *d*	• *t* = -1	•	•	•	-	-	-	-	-	-	F
Page *e*	-	-	-	-	-	-	F	•	•	-	-
Faults		•			•		•			•	•

Key take-aways from VMIN

- We can profitably unmap pages that are not likely to be used in the future
  - May accept more swapping for the current process, in exchange for less total swapping
- But, alas, we still don't have clairvoyance…
  - But we can use past behavior to guess…

Page Replacement Performance

- Local page replacement
  - LRU — Ages pages based on when they were last used
  - FIFO — Ages pages based on when they're brought into memory
- Towards global page replacement ... with variable number of page frames allocated to processes

The principle of locality

- 90% of the execution of a program is sequential
- Most iterative constructs consist of a relatively small number of instructions
- When processing large data structures, the dominant cost is sequential processing on individual structure elements
- Temporal vs. physical locality

The Working Set Model

- Assume recently referenced pages are likely to be referenced again soon…
- ... and *only* keep those pages recently referenced in memory (called *the working set*)
  - Thus pages may be removed even when no page fault occurs
  - The number of frames allocated to a process will vary over time
- A process is allowed to execute only if its working set fits into memory
  - The working set model performs implicit load control

Working Set Page Replacement

- Keep track of the last τ references (including faulting reference)
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, $\tau = 4$ references:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	• $t = 0$										
Page *b*	-										
Page *c*	-										
Page *d*	• $t = -1$										
Page *e*	• $t = -2$										
Faults											

Working Set Page Replacement

- Keep track of the last τ references
  - The pages referenced during the last τ memory accesses are the working set
  - τ is called the *window size*
- Example: Working set computation, $\tau = 4$ references:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• *t* = 0	•	•	•	-	-	-	-	-	F	•
	Page *b*	-	-	-	-	F	•	•	•	-	-	-
	Page *c*	-	F	•	•	•	•	•	•	•	•	•
	Page *d*	• *t* = -1	•	•	•	•	•	•	-	-	-	F
	Page *e*	• *t* = -2	•	-	-	-	-	F	•	•	•	•
Faults			•			•		•			•	•

Page-Fault-Frequency Page Replacment

- An alternate approach to computing working set
- Explicitly attempt to minimize page faults
  - When page fault frequency is high — *increase working set*
  - When page fault frequency is low — *decrease working set*

Algorithm:

Keep track of the rate at which faults occur

When a fault occurs, compute the time since the last page fault

Record the time, t_{last}, of the last page fault

If the time between page faults is "large" then reduce the working set

If $t_{current} - t_{last} > \tau$, then remove from memory all pages not referenced in $[t_{last}, t_{current}]$

If the time between page faults is "small" then increase working set

If $t_{current} - t_{last} \leq \tau$, then add faulting page to the working set

Page Fault Frequency Replacement

- Example, window size = 2
- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \leq 2$, just add faulting page to the working set

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	•										
	Page *b*	-										
	Page *c*	-										
	Page *d*	•										
	Page *e*	•										
Faults												
$t_{cur} - t_{last}$												

Page Fault Frequency Replacement

- Example, window size = 2
- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \leq 2$, just add faulting page to the working set

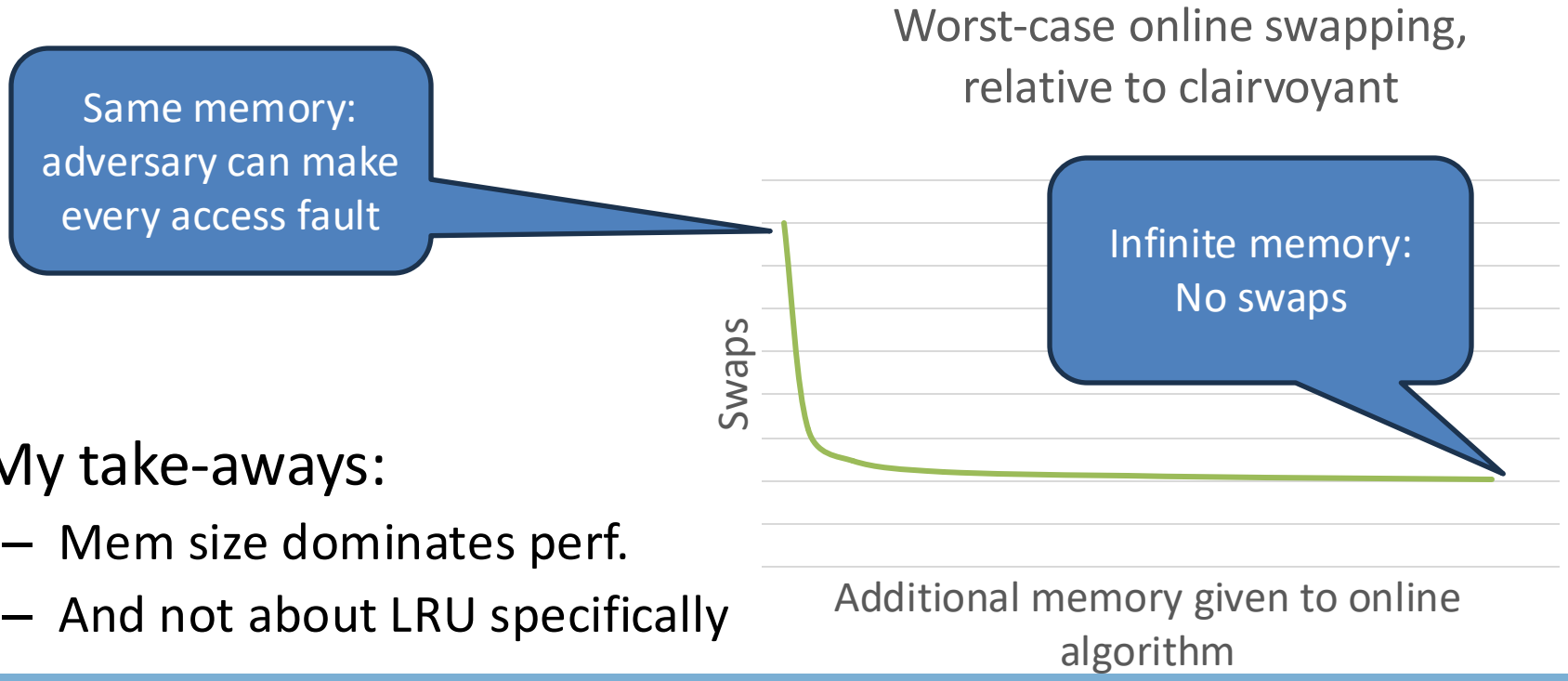
Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	•	•	•	•	-	-	-	-	-	F	•
	Page *b*	-	-	-	-	F	•	•	•	•	-	-
	Page *c*	-	F	•	•	•	•	•	•	•	•	•
	Page *d*	•	•	•	•	•	•	•	•	•	-	F
	Page *e*	•	•	•	•	-	-	F	•	•	•	•
Faults			•			•		•			•	•
$t_{cur} - t_{last}$			1			3		2			3	1

What does theory teach us?

- Famous result from Sleator and Tarjan's "Amortized Efficiency of List Update and Paging Rules", STOC 1985:
  - "We analyze the amortized complexity of LRU, showing that its efficiency differs from that of the off-line paging rule (Belady's MIN algorithm) by a factor that depends on the size of fast memory. No on-line paging algorithm has better amortized performance."
    - Online here just means no knowledge of the future (including observing prior executions)
    - And, although not stated here, excludes randomized algorithms
- Colloquially, sometimes taken to mean:
  - "LRU is optimal" for swapping

What the paper shows

- Their Theorem 5 shows a worst-case lower bound on any online algorithm (call it LRU) relative to clairvoyant
  - There is a trade-off space between how much extra memory LRU has vs how many additional swaps



- My take-aways:
  - Mem size dominates perf.
  - And not about LRU specifically

What the paper shows (2)

- Theorem 6 (paraphrased):
  - For LRU, FIFO, and some other non-terrible replacement algorithms the bound is tight
    - Can't do more than a constant worse than this
- My take-aways:
  - For non-brain-dead replacement algorithms, the previous bound is tight, within an additive constant
    - Brain-dead === LFU, LIFO, or ones that deliberately choose the opposite of recent history
  - Result not specific to LRU --- includes FIFO

Can you do better in practice?

- Sure!
- This is a worst-case analysis with absolutely no assumptions about program behavior, not an average case analysis

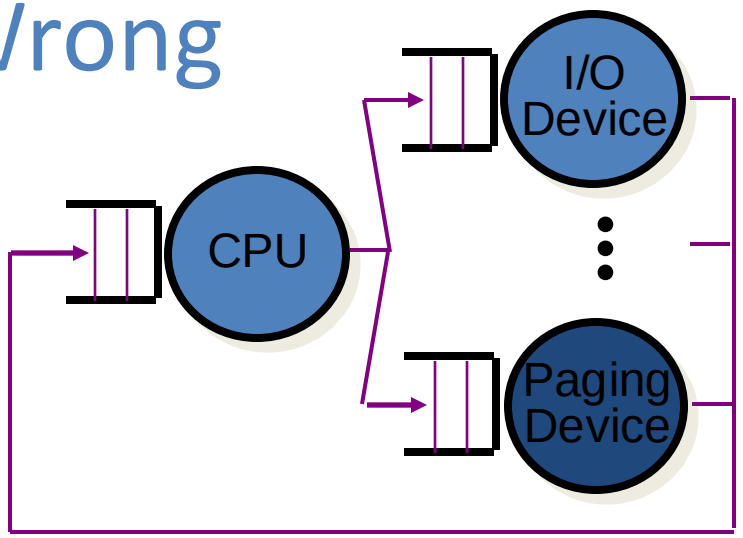
LRU not optimal for reduced associativity

- Another common misunderstanding of this result is that LRU is optimal for cache replacement in limited associativity.
  - Virtual memory is fully associative, CPU caches restrict placement of data to certain portions of the cache
- Recent work (Bender et al., under submission) proves that LRU is in fact not optimal
  - Some randomization actually improves performance compared to LRU by dealing with hotspots more effectively
  - Note: Shameless plug of my own research (I'm a coauthor)

Load Control: Fundamental Trade-off

- High multiprogramming level
  - $MPL_{max} = \frac{number\ of\ page\ frames}{minimum\ number\ of\ frames\ required\ for\ a\ process\ to\ execute}$
- Low paging overhead
  - MPL_{min} = 1 process
- Issues
  - What criterion should be used to determine when to increase or decrease the *MPL*?
  - Which task should be swapped out if the *MPL* must be reduced?

Load Control Done Wrong



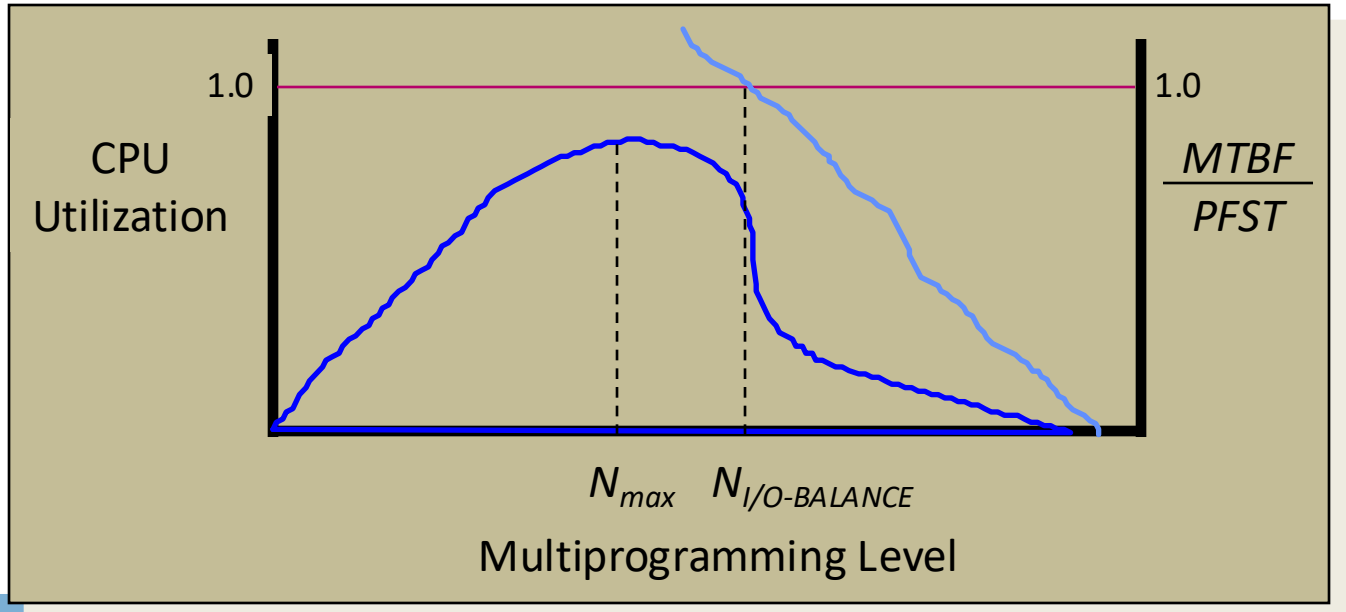
i.e., based on CPU utilization

- Assume memory is nearly full
- A chain of page faults occur
  - A queue of processes forms at the paging device
- CPU utilization falls
- Operating system increases *MPL*
  - New processes fault, taking memory away from existing processes
- CPU utilization goes to 0, the OS increases the *MPL* further...

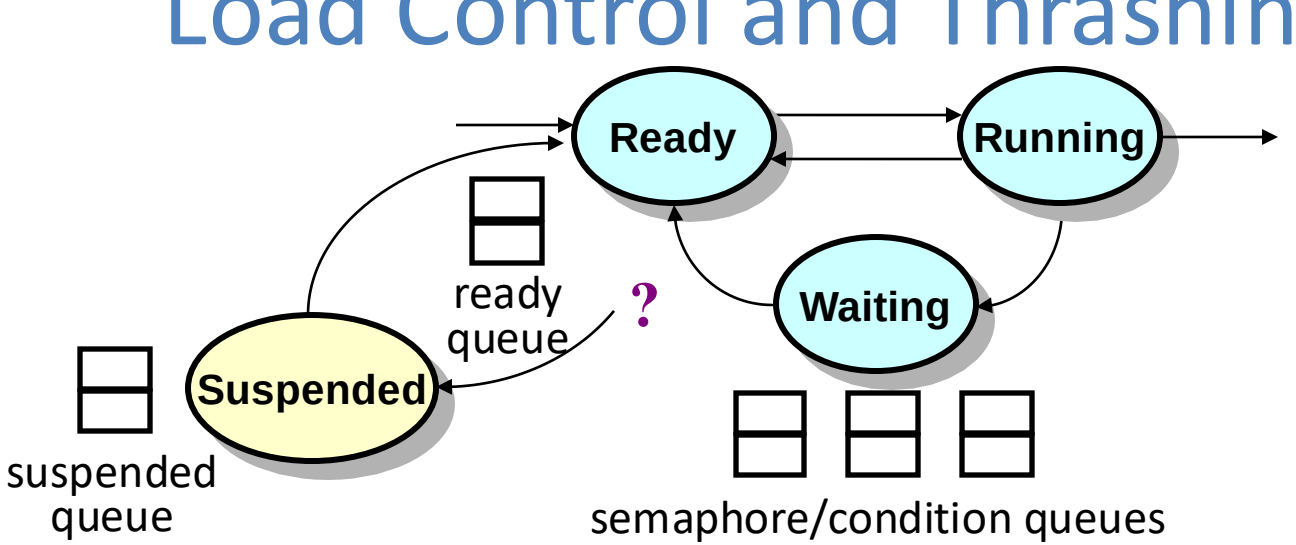
System is *thrashing* — spending all of its time paging

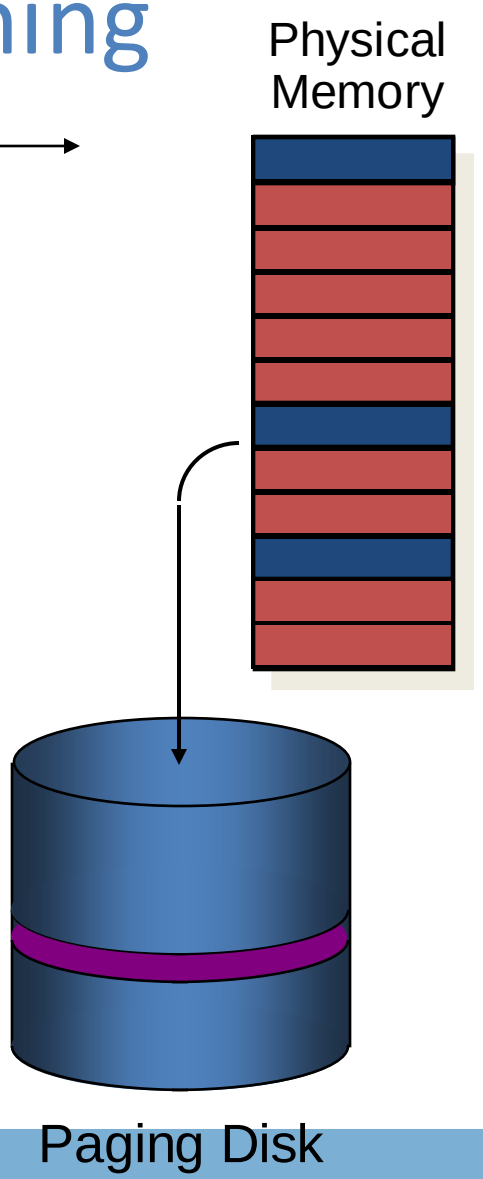
Load Control and Thrashing

- Thrashing can be ameliorated by *local* page replacement
- Better criteria for load control: Adjust MPL so that:
  - *mean time between page faults (MTBF) = page fault service time (PFST)*
  - $\Sigma\ WS_i$ *= size of memory*



Load Control and Thrashing





- When the multiprogramming level should be decreased, which process should be swapped out?
  - Lowest priority process?
  - Smallest process?
  - Largest process?
  - Oldest process?
  - Faulting process?